Défense nationale National Defence

UNCLASSIFIED

DEFENCE RESEARCH ESTABLISHMENT
CENTRE DE RECHERCHES POUR LA DÉFENSE
VALCARTIER, QUÉBEC

DREV-TM-9738

Unlimited Distribution/Distribution illimitée

ÉVALUATION D'UN SYSTÈME À RAISONNEMENT PROCÉDURAL (UM-PRS)

par

L. Lamontagne, M. Bélanger, J. Berger et P. Pelletier*

May/mai 1998

RESEARCH AND DEVELOPMENT BRANCH
DEPARTMENT OF NATIONAL DEFENCE
CANADA
BUREAU - RECHERCHE ET DÉVELOPPEMENT
MINISTÈRE DE LA DÉFENSE NATIONALE

Canada

SANS CLASSIFICATION

105-01893/

AVERTISSEMENT

Les renseignements ci-inclus sont la propriété de Sa Majesté et sont communiqués au destinataire, étant entendu qu'ils seront utilisés uniquement à des fins d'information et d'évaluation. Toute utilisation à des fins commerciales, notamment à des fins de fabrication, est interdite. La diffusion de la présente publication ou de son contenu à des tiers est interdite sans le consentement écrit préalable du MDN du Canada.

© Sa Majesté la reine, représentée par le ministre de la Défense nationale, 1998

UNCLASSIFIED

DEFENCE RESEARCH ESTABLISHMENT

CENTRE DE RECHERCHES POUR LA DÉFENSE

VALCARTIER, QUÉBEC

DREV-TM-9738

Unlimited Distribution/Distribution illimitée

ÉVALUATION D'UN SYSTÈME À RAISONNEMENT PROCÉDURAL (UM-PRS)

par

L. Lamontagne, M. Bélanger, J. Berger et P. Pelletier*

May/mai 1998

*APG Inc.

Approved by/Approuvé par

Gaétan Picard

Section Head/Chef de section

13/5/98

Date

SANS CLASSIFICATION

ABSTRACT

Even though command and control (C2) systems are deployed in non-deterministic environments with rapidely changing and uncertain conditions, very few mechanisms supporting these characteristics are incorporated in their design. The "Adaptive Intelligent System" (AIS) paradigm offers solutions that could enhance C3I systems response to these deficiencies. For example, AIS technologies enable the adaptation of system behavior to resource limitations, temporal deadlines and other limiting factors inherent to the environment. This document describes the principles of one of these paradigms, the Procedural Reasoning System (PRS). Furthermore, an evaluation of a PRS implementation, UM-PRS, a software package developed by a research team at the University of Michigan, is presented.

RÉSUMÉ

Bien que les systèmes de commandement et contrôle soient déployés dans des environnements non déterministes dont les conditions évoluent rapidement et de manière incertaine, peu de mécanismes adaptés à ces caractéristiques sont utilisés lors de leur conception. Le paradigme des "systèmes adaptatifs intelligents" (AIS) offre des solutions qui pourraient pallier ces lacunes; il permet, entre autres, d'adapter le comportement d'un système en réponse à des limitations de ressources, des échéances temporelles et d'autres facteurs contraignants liés à l'environnement. Dans ce document, on décrit les principes de l'un de ces paradigmes, le système de type "à raisonnement procédural" ("Procedural Reasoning System", PRS). On présente également l'évaluation de UM-PRS, un système développé par une équipe de chercheurs de l'Université du Michigan.

TABLE DES MATIÈRES

ABSTRACT I

FICHE DE SYNTHÈSE IV

1.0 INTRODUCTION 1

2.0 PRS : PRINCIPES ET ARCHITECTURE 2

2.1 L'architecture du modèle original 2

2.2 L'interpréteur et la boucle de contrôle 6

2.3 Systèmes à raisonnement procédural existants 7

3.0 DESCRIPTION DU SYSTÈME UM-PRS 8

3.1 Le langage de description 9

3.2 Commandes permises dans le corps d'un KA 11

3.3 Environnement de développement 13

3.4 Primitives 14

4.0 IMPLANTATION DU SYSTÈME UM-PRS 15

4.1 Quelques aspects généraux 15

4.2 Les classes du système 16

5.0 PERFORMANCES DU SYSTÈME UM-PRS 18

5.1 Aspects à considérer qui influencent la performance du système 18

5.2 Résultats des tests 21

6.0 CONCLUSION 23

7.0 RÉFÉRENCES 25

ANNEXE A - Description des fichiers du système UM-PRS 26

ANNEXE B - Description des classes du système UM-PRS 27

ANNEXE C - Solutions proposées pour les évènements déclencheurs.....42

FIGURES 1 À 10

TABLEAUX I À VII

FICHE DE SYNTHÈSE

Bien que les opérations de commandement et contrôle soient menées dans des environnements dont les conditions évoluent rapidement et de manière incertaine, les systèmes utilisés lors de ces opérations s'adaptent difficilement à ces changements; leur capacité de traitement étant limitée et leur mécanisme de contrôle faisant abstraction des variations dans l'environnement. Le paradigme des "systèmes adaptatifs intelligents" (AIS) pourrait offrir des solutions pour pallier ces lacunes. Les technologies AIS permettent, entre autres, d'adapter le comportement d'un système en réponse à des limitations de ressources, des échéances temporelles et d'autres facteurs contraignants.

L'objectif de cette étude était d'évaluer le potentiel des systèmes à raisonnement procédural ("Procedural Reasoning Systems", PRS) comme infrastructure pour un système adaptatif intelligent. Ces systèmes sont qualifiés de "procéduraux" parce qu'ils s'appuient sur une connaissance structurée (similaire à des procédures) sous forme de graphe dirigé. Leur mécanisme de contrôle, qui combine la direction par les buts et les nouvelles données, offre la flexibilité nécessaire à la construction de mécanismes de type adaptatif intelligent.

Ce document présente les résultats de l'étude du système UM-PRS (Réf. 6), l'implantation du modèle PRS que nous avons retenu. Nous décrivons les composantes, la mise en oeuvre et l'évaluation de certaines propriétés du système UM-PRS. Etant donné les résultats de l'évaluation, nous avons décidé de ne pas opter pour UM-PRS pour les raisons suivantes: la base de données s'appuie sur un formalisme de représentation de connaissance limité, la performance de la création de nouveaux faits est non linéaire et le système ne comporte pas de couche de méta-niveau (méta-KA ou autre concept similaire).

1.0 INTRODUCTION

Bien que les opérations de commandement et contrôle soient menées dans des environnements dont les conditions évoluent rapidement et de manière incertaine, les systèmes de commandement et contrôle actuels s'adaptent difficilement à ces changements; leur capacité de traitement étant limitée et leur mécanisme de contrôle faisant abstraction des variations dans l'environnement.

Le paradigme des "systèmes adaptatifs intelligents" (AIS) offre des solutions pour pallier ces lacunes. Un système adaptatif intelligent est un système qui adapte son comportement interne en fonction des variations dans l'environnement externe. Les technologies AIS permettent, entre autres, de s'adapter en fonction de limitations des ressources, d'échéances temporelles et de changement de buts/tâches dictés par l'usager du système.

Un projet est mené, depuis février 1996, par un groupe de chercheurs du Centre de recherches pour la défense Valcartier (CRDV), afin d'étudier les propriétés des systèmes adaptatifs intelligents. Ces études permettront de spécifier et concevoir une architecture AIS pouvant être utilisée pour des applications de commandement et contrôle.

Ce document décrit la portion du projet dont l'objectif était d'évaluer le potentiel des systèmes à raisonnement procédural ("Procedural Reasoning Systems", PRS) comme infrastructure d'un système adaptatif intelligent. Ces systèmes sont qualifiés de "procéduraux" parce qu'il s'appuient sur une connaissance structurée (similaire à des procédures) sous forme de graphe dirigé. Leur mécanisme de contrôle (raisonnement) sélectionne et exécute des plans prédéterminés (procédures) en réponse à des buts à atteindre et de nouvelles données. Par rapport aux objectifs de notre étude, ce type de contrôle présente de l'intérêt, puisqu'il offre la flexibilité nécessaire à la construction de mécanismes de type adaptatif intelligent.

Les principes de base des systèmes à raisonnement procédural sont présentés au chapitre 2.0. Nous décrivons au chapitre 3.0 les composantes d'un de ces systèmes, UM-PRS, et présentons sa mise en oeuvre au chapitre 4.0 et à l'annexe B. Finalement, nous présentons l'évaluation de certaines propriétés de ce système (chapitre 5).

Ce travail a été exécuté au CRDV entre janvier et mars 1996 dans le cadre de l'unité de travail 3aa15, "Planification dynamique et exécution".

2.0 PRS : PRINCIPES ET ARCHITECTURE

Les systèmes à raisonnement procédural (PRS) ont pour origine les travaux de recherche de Georgeff et Lansky (Réf. 1) sur la représentation de connaissances procédurales. Dans ce formalisme, une procédure, conçue pour être exécutée dans un environnement complexe et dynamique, est une description explicite d'un plan d'actions prédéfinies.

L'architecture d'un système PRS (décrite à la section 2.1) permet l'exécution de ce type de procédures. Son moteur d'inférence (décrit à la section 2.2) est basé sur les notions de but et d'événement et permet l'intégration de comportements de réaction et de planification. Cette intégration, couplée à la possibilité de gérer concurrement plusieurs tâches, fait du système PRS une architecture particulièrement bien adaptée aux systèmes intelligents adaptatifs. De plus, cette architecture est particulièrement bien appropriée aux domaines où il existe des procédures prédéfinies pour réagir à différentes situations. Ainsi, un tel système pourrait appuyer l'utilisation, dans les postes de commandement militaires, de procédures décrites sous forme de "Standard Operating Procedures".

2.1 L'architecture du modèle original

Une architecture de type PRS (Réf. 2) comporte quatre bases de connaissances (fig. 1): une base de données, une structure de buts, un ensemble de "Knowledge Areas"

(KA) et une structure d'intentions. Ces composantes sont décrites aux paragraphes suivants. La cinquième composante, l'interpréteur, est décrite au chapitre 2.2.

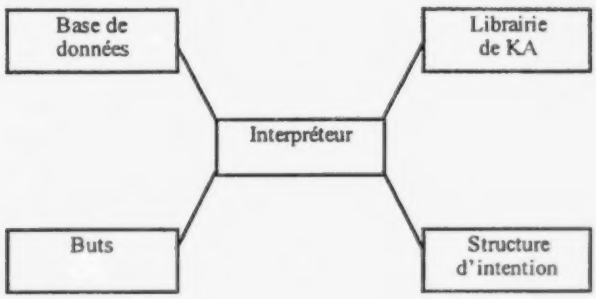


FIGURE 1 - Architecture d'un système à raisonnement procédural

2.1.1 Base de données ("World Model")

La base de données (BD) contient la description du monde, *i.e.* des faits et croyances que le système "connaît" sur l'environnement dans lequel il oeuvre. Ces connaissances portent aussi bien sur les propriétés statiques d'un domaine d'application (par ex. les limites physiques d'un avion) que sur les données acquises ou dérivées par le système en opération (connaissances dynamiques telles que la disponibilité d'un avion à un moment donné). De plus, la base de données peut contenir une description de l'état interne du système (ex: nombre de cycles effectués par l'interpreteur).

Le contenu de la base de données est décrit à l'aide d'une syntaxe de logique du premier ordre. Par exemple, selon ce type de représentation, l'expression

(located aircraft1 location1)

signifie que l'avion identifié "aircraft1" est localisé à la position "location1".

Dans le modèle initial, la base de données est indexée par termes ("term indexing") pour donner un temps d'accès constant aux connaissances. Ce temps d'accès constant est un facteur particulièrement important pour garantir la réactivité du système en temps quasi-réel.

2.1.2 Structure de buts ("Goal structure")

La structure de buts est une liste constituée de buts, i.e. des expressions décrivant les conditions à satisfaire lors de l'exécution du système. Ces conditions sont composées de deux éléments: un opérateur temporel et une description d'état (interne ou externe au système). Les opérateurs définis dans le modèle initial permettent de vérifier, d'atteindre, de maintenir ou d'éviter un état. Les symboles utilisés sont respectivement: ?, !, ^ et ¬. Les états sont décrits de la même façon que les faits de la base de données (*i.e.* selon une logique du premier ordre). Par exemple, l'expression

! (deployed airforce1 airfield2 end-time1)

signifie que l'on veut que l'unité "airforce1" soit déployée à la base aérienne "airfield2" au temps "end-time1" (*i.e.* **atteindre** l'état "airforce1 déployé à "airfield2" au temps "end-time1"). Par contre, l'expression

? (deployed airforce1 airfield2 end-time1)

signifie que l'on veut **vérifier si** l'unité "airforce1" est déployée à la base aérienne "airfield2" au temps "end-time1".

Les buts, utilisés pour guider le mécanisme de contrôle du système PRS, sont soit fournis par l'utilisateur lors de l'initialisation du système, soit créés au moment de l'exécution des procédures (KA).

2.1.3 Ensemble de "Knowledge Area" ("KAs Library")

Cette ensemble contient une liste des procédures, chacune étant appelée "Knowledge Area" (KA). Un KA est une forme de procédure ou de plan décrivant une suite d'actions exécutables par le système. Il se caractérise par (voir exemple à la fig. 2):

- la condition d'invocation ("purpose"), décrivant le but qu'il permet de satisfaire **ou** la situation à laquelle il permet de réagir;

- le contexte ("context") où le KA est applicable; et
- le corps, *i.e.* les étapes de la procédure. Le corps est structuré comme un graphe dont les liens décrivent les sous-buts à être satisfaits pour compléter l'exécution de ce KA. Ainsi, l'exécution d'un KA peut amener l'exécution d'autres KA, appelés sous-KA.

FIGURE 2 - Exemple de "Knowledge Area"

Les KA, définis par le concepteur du système, peuvent être de deux types: ceux dont le corps comporte uniquement des actions modifiant l'environnement externe et la base de données du système sont appelés tout simplement KA et ceux dont le corps comporte des actions modifiant le comportement de la boucle de contrôle du système sont appelés Méta-KA. L'utilisation des méta-KA est abordée à la section 2.2.

2.1.4 Structure d'intentions ("Intention Structure")

La structure d'intentions contient toutes les intentions, *i.e.* les tâches qui ont été sélectionnées par le système pour être exécutées immédiatement ou à un moment ultérieur. Une intention contient le KA initial, tous les sous-KA selectionnés pour compléter l'exécution du KA initial, ainsi que l'expression du but ou du fait qui a provoqué son invocation.

2.2 L'interpréteur et la boucle de contrôle

L'interpréteur guide l'exécution du système PRS (boucle de contrôle, interfaces, etc.). Pour les fins de notre étude, nous nous intéresserons uniquement à la boucle de contrôle (dont le pseudo-code est présenté à la fig. 3). Nous pouvons résumer le cycle de la boucle de contrôle comme suit: à la suite de la création de nouveaux buts ou de nouveaux faits, un sous-ensemble de KA applicables est identifié et conservé sous forme de "Set Of Applicable KA" (SOAK); quelques éléments de ce SOAK sont sélectionnés et déposés dans la structure d'intention; finalement une intention est choisie et exécutée.

```
while(TRUE) {
        CHECK_INPUT_BUFFER();
        GET_NEW_FACTS_GOALS();
        soak = FIND_APPLICABLE_KAS();
        while (! (LIST_EMPTY(soak))) {
                POST_SOAK_META_FACTS(soak);
                previous_soak = soak;
                GET_NEW_FACTS_GOALS();
                soak = FIND_APPLICABLE_KAS();
        }
        if (! (LIST_EMPTY(previous_soak))) {
                POST_SOAK_META_FACTS(soak);
                intend(SELECT_RANDOMLY(previous_soak));
        }
        current_intention = CHOOSE_INTENTION();
        EXECUTE_INTENTION(current_intention);
        previous_soak = soak;
}
```

FIGURE 3 - Boucle de contrôle FRS (en pseudo-code)

Dans le modèle initial, le cycle de la boucle de contrôle est répété indéfiniment mais pour d'autres systèmes de la famille PRS, cette boucle peut être interrompue dans différentes conditions, telles que l'absence de buts dans la structure de buts, un SOAK vide, ou l'occurrence de certains événements externes.

Les méta-KA peuvent être utilisés pour contrôler chacune des étapes de la boucle de contrôle: déduction de nouvelles connaissances à partir de nouvelles données, construction ou modification du SOAK, sélection des KA pour l'exécution, l'ordonnancement de l'exécution des KA par la réorganisation de la structure d'intention, l'interruption de l'exécution d'un KA. Un exemple de méta-KA, décrit en pseudo-code, est présenté à la fig. 4.

```
KA {
    PURPOSE:
            (soak $list_of_kas);
    CONTEXT:
            (> (length $list_of_kas) 1);
    BODY:
            $ka_activated_par_event = (FACT_INVOKED_KA $list_of_kas);
            when (? (> (length $ka_activated_by_event) 0)) {
                EXECUTE_INTEND_FACT $ka_activated_by_event;
            }
            when (? (= (length $ka_activated_by_event) 0)) {
                EXECUTE_INTEND_GOAL_RANDOM $ka_activated_by_event;
            }
}
```

FIGURE 4 - Exemple de Méta-KA

2.3 Systèmes à raisonnement procédural existants

À la connaissance des auteurs de ce document, il existait, en 1996, trois (3) implantations de systèmes à raisonnement procédural: CL-PRS, C-PRS et UM-PRS.

Le CL-PRS est un système qui a été développé au SRI de Stanford (Réf. 2) à partir du modèle initial de Georgeff et Lansky (Réf. 1). Il est programmé en Common Lisp et n'est pas disponible, car l'équipe présentement en charge de son développement l'utilise à des fins de consultation. Le CL-PRS est également utilisé pour des travaux de recherche; par exemple, un formalisme a été défini pour intégrer cette architecture à un système de génération de plan (SIPE-2) (Réf.3).

Le C-PRS (Réfs. 4 et 5) est une version commerciale de système à raisonnement procédural. Elle a été développée par la firme française ACS Technologies en collaboration avec le chercheur Francois Félix Ingrand. Le logiciel est développé en langage C et offre de nombreux outils, comme un éditeur graphique de KA, des modules de communications informatiques et un interface usager. Le coût du logiciel, en 1996, était d'environ 125 000 FF (donc approximativement $CAD 30 000).

L'UM-PRS (Réfs. 6 et 7) est un logiciel expérimental développé par le laboratoire d'intelligence artificielle de l'Université du Michigan. Il est programmé en C++ standard et ne nécessite l'utilisation d'aucun autre logiciel. Nous l'avons obtenu gratuitement pour fins de recherche du Dr. Edmond Durfee (directeur du laboratoire). La description du

système UM-PRS est présentée au chapitre 3.0 de ce document. La mise en oeuvre orientée-objet du système est décrite au chapitre 4.0 et à l'annexe B. Finalement, une évaluation de quelques-unes des ses fonctionnalités est présentée au chapitre 5.0

3.0 DESCRIPTION DU SYSTÈME UM-PRS

Le système UM-PRS se conforme en grande partie au modèle initial (chapitre 2.0), mais comporte quelques différences, dont trois fondamentales:

- **événement déclencheur:** Le mécanisme de déclenchement de KA à partir d'événements (nouvelles données, nouveaux faits) n'est pas intégré à l'implantation de l'UM-PRS. Ainsi, seule la création de nouveaux buts permet d'activer un KA, indépendamment des modifications apportées aux bases de connaissances (structure d'intentions, base de données, etc.). À partir de discussions que nous avons eues avec les concepteurs du système, des solutions ont été identifiées pour simuler le mécanisme d'événements déclencheurs, soit par l'utilisation de scripts, soit en modifiant légèrement le noyau de l'UM-PRS. Ces solutions sont présentées à l'annexe C intitulée ¨Solutions proposées pour l'implantation d'événements déclencheurs.
- **méta-KA:** Il n'existe pas de méta-KA dans l'UM-PRS. Dans le modèle initial, les méta-KA étaient destinés à modifier le processus de sélection d'intentions à partir du SOAK. Les concepteurs d'UM-PRS ont préféré remplacer ce mécanisme de sélection par un ordonnancement basé sur des valeurs des priorités (voir point suivant). Ainsi, l'élément du SOAK (l'intention) dont la priorité, déterminée à partir des priorités des buts et de KA, est la plus élevée sera présenté à la structure d'intention, comme sélection finale.
- **intention:** Comme le concept d'événement déclencheur n'existe pas, l'intention est seulement constituée d'une description de but et d'un KA applicable. De plus, la priorité par défaut d'une intention est définie comme la somme des priorités du KA et du but associé à l'intention. Par contre, le

concepteur a la possibilité de définir d'autres opérateurs afin de combiner les priorités.

3.1 Le langage de description

Le langage de description de l'UM-PRS permet à l'usager de définir les faits, les buts et les KA propres à une application. Ce langage est assez puissant pour écrire des KA ayant des commandes de boucles de branchements, de création de variables, de génération de buts, etc. (voir section 3.2). Pour exécuter des actions plus complexes ou de plus bas niveau, le concepteur peut ajouter des primitives, programmées en C++, au noyau de base de l'UM-PRS (voir section 3.4).

Un fichier de description, écrit à l'aide du langage de description, est constitué des quatre sections suivantes: "GOALS:", "FACTS:", "CYCLE:" et "KA". L'ordre de présentation des sections n'est pas important et chacune peut apparaître plus d'une fois.

La section "GOALS:" comprend les buts à satisfaire, tels que présentés à l'exemple suivant:

```
GOAL:
        ACHIEVE: Initilize ;
        ACHIEVE: ReachPointC "Rocket1" 10;
        ACHIEVE: Terminate;
```

La section "FACTS:" décrit les faits constituant l'état initial de la base de données. Un fait peut contenir des primitives qui seront évaluées lors de la consultation.

```
FACTS:
        test_done 0;
        position 42 13;
        temperature_fournaise (lire_temperature "fournaise");
                                              // appel de primitive
```

La section "CYCLE:" contient les KA et les fonctions qui seront exécutées à chaque cycle de l'interpréteur (voir exemple suivant).

```
CYCLE:
{
        RETRIEVE cycle_count $count;
        UPDATE cycle_count (+ $count 1);
}
```

La section "KA" décrit chacun des attributs d'un KA (NAME, DOCUMENTATION, PURPOSE, CONTEXT, BODY et FAILURE). Les commentaires de l'exemple suivant illustrent la signification de chacun de ces attributs. Un exemple de section KA est présenté à la fig. 5.

Un point à considérer lors la description d'un KA est son contexte et sa réévaluation avant l'exécution de chacune des instructions du KA. Une modification de contexte peut rendre invalide à tout moment le cours d'exécution d'un KA.

```
KA (                                    Débute la définition d'un KA.
   NAME:                                Nom du KA (purement descriptif puisqu'il n'est utilisé
     "mo-KA";                           que pour fin de "debuggage").

   DOCUMENTATION:                       Optionnelle. La section "DOCUMENTATION:" sert à décrire
     "Ceci est une description";        le KA.

   PURPOSE:                             La section "PURPOSE:" est constituée d'une expression
     ACHIEVE: Initialize;               de type BUT que le KA peut satisfaire

   CONTEXT:                             Optionnelle. La section "CONTEXT:" contient une liste
      FACT system_initialized           de conditions (le contexte) qui déterminent si le KA
"FALSE";                                est applicable en fonction du contenu de la BD. S'il
      FACT current_system               n'est pas défini, le contexte est considéré comme
"Alpha";                                toujours applicable.

   BODY:                                La section "BODY:", le corps du KA, contient la série
     UPDATE system_initialized          d'actions à exécuter. Cette section peut contenir les
"TRUE";                                 commandes décrites à la section 3.2. Les actions sont
                                        exécutées jusqu'à ce qu'une échoue, ou jusqu'à ce qu'on
                                        atteigne la dernière action du KA.

   FAILURE:                             Optionnelle. La section FAILURE: est une liste d'action
     EXECUTE print "Ooops";             à exécuter si le KA échoue en cours d'exécution. Cette
)                                       action est exécuté en mode ATOMIC, c'est-à-dire sans
                                        redonner le contrôle à l'interpréteur entre chacune des
                                        actions de sa liste.
```

FIGURE 5 - Section "KA" d'un fichier de description

Les types de donnée admissibles pour les arguments utilisés dans la description des différentes sections sont les suivants:

entier: correspond à un "*long*" en C++;

Réel: correspond à un "*double*" en C++;

string: chaîne de caractères;

pointeur: type générique pouvant contenir l'adresse de tout type de données;

variable: constituée d'un $ suivi d'un symbole. Par exemple, la variable $x peut prendre comme valeur les types définis ci-dessus.

3.2 Commandes permises dans le corps d'un KA

Les commandes pouvant être utilisées dans le corps d'un KA permettent de manipuler la base de données (tableau I), de manipuler la structure de buts (tableau II), de définir différents cours d'actions possible (tableau III), de boucler sur différentes actions (tableau IV) et d'exécuter des actions en mode ATOMIC (tableau V).

TABLEAU I

Commandes de manipulation de la base de données

ASSERT	*nom_de_predicat liste_de_faits*	Ajoute une connaissance à la BD. Ex: *ASSERT: position 42 10;*
RETRACT	*nom_de_predicat liste_de_faits*	Enlève une connaissance ou groupe de connaissances de la BD. Ex: *RETRACT position 10 42;* *RETRACT position;* i.e. retirer les faits ayant comme prédicat "position"
UPDATE	*(fait1)* *(fait2)*	Remplace un fait par un autre. Ex: *UPDATE: (position 10 11) (position 20 42);* *UPDATE: (position) (position 20 42);* i.e. remplacer tous les faits ayant comme prédicat *position.*
FACT	*nom_de_predicat liste_de_faits_ou_variables*	Teste la présence du fait dans la BD. Utile pour l'instanciation de variables. Ex: *FACT position 10 10;* i.e. test *FACT position $x $y;* i.e. instancie $x et $y à partir du contenu de la BD.

TABLEAU II

Commandes de manipulation de la stucture de buts

POST	*expression_de_but*	Ajoute un but principal (sans but parent) sur la structure de buts. Ex: *POST: ACHIEVE Goal1 10 32;*
UNPOST	*expression_de_but*	Enlève un but dans la structure de buts. Ex: *UNPOST: ACHIEVE Goal1 10 32;*
ACHIEVE	*nom_de_but* [*arguments*] [:PRIORITY *expression*] [:BY *noms_de_KA*] [:NOT-BY *noms_de_KA*]	Ajoute un sous-but (généré par un autre but) dans la structure de buts. ACHIEVE peut être suivie d'une priorité. Ex: *ACHIEVE: Goal1 :PRIORITY 10;* Les commandes de buts peuvent être suivies de :BY et/ou :NOT-BY qui indiquent les KA à retenir ou ne pas retenir lors de la sélection. Ex: *ACHIEVE Goal1 :BY "Ka-1" "Ka-2" $ka_name :NOT-BY "Ka-3";*
WAIT	*expression_de_but*	Suspend l'exécution d'un KA jusqu'à ce que le but spécifié soit satisfait.(en cours d'implantation).
MAINTAIN	*expression_de_but*	Tente d'atteindre le but à nouveau si celui-ci échoue (en cours d'implantation).

TABLEAU III

Commandes de branchement

TEST	*expression*	Commande qui évalue l'expression et réussit si le résultat est vrai, non nul ou une chaîne de caractère non vide. Ex: *TEST (< $count 20);*
OR	*bloc_d'actions*	Exécute les blocs de commandes jusqu'à ce que l'un réussisse. S'ils échouent tous, la commande OR échoue. Ex: *OR* *{ ACHIEVE Goal1; }* *{ ACHIEVE Goal2; };*
AND	*bloc_d'actions*	Exécute tous les blocs de commandes jusqu'à ce que l'un échoue. S'ils réussissent tous, la commande AND réussit, sinon elle échoue. Ex: *AND* *{ TEST (== "a1" "b1"); }* *{ TEST (== "a2" "b2"); };*
WHEN	*test* *bloc_d'actions*	Exécute un bloc d'actions si le test est satisfaisant. Ex: *WHEN : TEST (> $x 10)* *{ ACHIEVE goal1;* *UPDATE (test_done) (test_done "TRUE");* *}*

TABLEAU IV

Commandes de boucle

WHILE	*test* *bloc_d'actions*	Exécute un bloc d'actions jusqu'à ce que le TEST échoue. Ex: *WHILE : TEST (> $count 10)* *{ ASSIGN $count (+ $count 1);* *}*
DO	*bloc_d'actions* *testf*	Exécute un bloc de commandes jusqu'à ce que le TEST échoue. Ex: *DO:* *{ASSIGN $count (+ $count 1);}* *WHILE: TEST (> $count 10);*

TABLEAU V

Commandes atomiques

ATOMIC	*bloc_d'actions*	Exécute une série d'actions comme s'il ne s'agissait que d'une seule, i.e. sans interruption de la part de l'interpréteur. Ex: *ATOMIC* *{UPDATE (debug_mode) (debug_mode "FALSE");* *UPDATE (run_mode) (run_mode "TRUE");* *}*
EXECUTE	*nom_primitive* *arguments*	Exécute la primitive. Cette primitive peut être prédéfinie dans le noyau de l'UM-PRS ou une définie par le concepteur. Ex: *EXECUTE get_soak_element $Soak $index $SoakElement;*

Il n'existe aucune commande pour la manipulation du SOAK ou de ses éléments (SoakElements). De telles commandes pourraient être implantées sous forme de primitives, mais la grammaire actuelle du langage est peu robuste et se prête mal à cet exercice; la grammaire telle que définie en YACC permet une définition de définition erronée pouvant rendre le système instable.

3.3 Environnement de développement

Il n'existe pas d'environnement de conception intégré à l'UM-PRS. Le système est un fichier exécutable qui fait la lecture du fichier de description et qui l'exécute. Tel que mentionné précédemment, des environnements de travail ont été créés pour d'autres systèmes à raisonnement procédural tels que C-PRS (Réf. 5) et CL-PRS (Réf. 3). Il serait intéressant de doter l'UM-PRS d'un environnement de travail permettant la visualisation des KA sous forme graphique et d'un outil pour l'inspection des différents structures du

système, telles que la structure d'intention et la base de données. Aucun effort n'a été consacré à cette activité, puisqu'elle ne correspondait pas aux objectifs de notre étude.

3.4 Primitives

Une des forces de UM-PRS est la possibilité de définir de nouvelles fonctions et ainsi augmenter le noyau du système. Des primitives (macros et fonctions prédéfinies par les concepteurs d'UM-PRS) sont disponibles pour faciliter la création de nouvelles fonctions. Ces primitives sont présentées au tableau VI. Une portion du langage de commandes d'UM-PRS est d'ailleurs définie à l'aide de ces primitives.

La création de nouvelles primitives se fait à l'aide d'une macro définie dans le fichier "primitive.h" comme suit:

```
#define PRIMITIVE_FUNCTION(name) Value name(long arity, ExpList* args, Binding*
binding = 0)
```

Une primitive est une fonction recevant comme arguments:

- le nombre d'arguments de la primitive ("arity"),
- la liste des arguments de la primitive,
- l'instanciation ("*bindings*") des variables du système, *i.e.* la liste des valeurs attribuées aux variables.

Lorsque l'exécution d'une primitive réussi, la valeur obtenue est *Value::True* et lorsqu'elle échoue *Value::False*.

Les arguments de la primitive peuvent être instanciés ou non, et il est possible de passer des résultats par ceux-ci. Les arguments de type "variable" ne sont pas instanciés. Le "binding" doit alors être utilisé explicitement par le concepteur afin des les instancier.

```
Ex:     uneVariable.eval(binding);
```

TABLEAU VI

Primitives définies dans UM-PRS

+, -, *, /, %	Opérations arithmétiques
==, !=, <, <=, >, >=, and, &&, or, ||, not !	Opérations logiques
Print *liste_d'arguments*	Fonction affichant la liste des arguments à l'écran
Create_relation *nom_de_relation liste_d'arguments*	Fonction ajoutant un fait à la BD (les arguments doivent être instanciés)
Sleep *nombre_de_secondes*	Fonction suspendant l'application pendant un certain nombre de secondes
read_integer *liste_de_variables*	Attribut aux variables des entiers entrés au clavier
read_float *liste_de_variables*	Attribut aux variables des nombres réels entrés au clavier
read_string *liste_de_variables*	Attribut aux variables des chaînes de caractères entrées au clavier
generate_soak, print_soak *et* **delete_soak**	Primitives de manipulation du SOAK
remove_ka *nom_de_ka*	Focntion permettant le retrait d'un KA de la librairie
abs *un_nombre*	Fonction retournant la valeur absolue d'un nombre
get_time *variable*	Fonction instanciant la valeur de la variable à l'heure du système
shell *liste_d'arguments*	Fonction exécutant les commandes Unix données en argument

4.0 IMPLANTATION DU SYSTÈME UM-PRS

L'implantation du système UM-PRS, programmé en C++, est standard et ne fait appel à aucune librairie spécifique à UNIX, ce qui rend le système facilement portable sur différentes plates-formes. Le code est assez bien structuré et clair. Il est donc relativement facile d'apporter des modifications au système de base. Une description des fichiers du système est présentée à l'annexe A.

4.1 Quelques aspects généraux

Les différentes bases de connaissances du système sont déclarées au niveau global et sont donc accessibles par tous les objets d'UM-PRS. Par contre, le système n'offre aucun langage de requête pour les connaissances du système; la création de primitives de manipulation de la structure de buts, de l'ensemble de KA et de la structure d'intentions

apporterait une plus grande souplesse au syst me UM-PRS. De plus, elle permettrait de créer des "méta-ka"; ce qui rendrait plus attrayante l'utilisation de scripts.

La création de nouvelles primitives se fait aisément; ceci permet d'ajouter des fonctionnalités additionnelles au syst me UM-PRS sans en modifier le noyau. Par ailleurs, ensemble de primitives plus étoffée permettrait la création de scripts (primitives) de plus haut niveau. Dans la version actuelle du système, le concepteur doit lui-même créer un bon nombre de primitives qui devraient logiquement se trouver dans le système de base.

4.2 Les classes du système

Les principales classes d'UM-PRS, ainsi que les relations entre elles sont présentées à la fig. 6. Une description générale de ces principales classes est présentée au tableau VII; une description plus détaillée de classes, d'attributs, de méthodes et de quelques algorithmes est présentée à l'annexe B.

TABLEAU VII

Description générale des principales classes d'UM-PRS

Program	Cette classe contient les principaux éléments d'UM-PRS, ainsi que les options sélectionnées lors de son exécution. Une instance de la classe Program, *program*, est créée lors de l'exécution et est visible globalement par toutes les autres composantes.
Relation	Un élément de connaissance. Il associe un fait à une liste de valeurs.
WmTable (WorldModel)	Une spécialisation de SymbolTable, qui est une table de hashing. Cette classe gère la base de données et est composée de Relations. Une relation peut être retrouvée avec son nom ou, pour être plus performant, son identifiant (ID), qui est le résultat de la fonction de hashing. L'ID est le moyen le plus souvent utilisé par les autres classes pour conserver un lien avec une Relation.
Goal	La classe décrivant la notion de but. Des instances de cette classe sont insérées dans la structure de buts soit au moment de l'initialisation du système, dynamiquement avec la commande POST ou en exécutant un sous-but.
GoalSet	Elle gère la structure de buts et offre des services pour la manipuler.
Ka	Elle implante la notion de KA telle que décrite précédemment Ses services sont surtout reliés aux conditions d'activation. L'exécution du KA est géré par la structure d'intention, les services du KA à ce sujet se limitant à créer un "run-time frame" approprié pour l'action courante.
KaBodyElement	L'élément de base dont est constitué le corps d'un KA. Les KaBodyElement sont groupés sur une liste chaînée constituant les actions à exécuter. On peut les classer en

	deux groupes: les éléments simples et les éléments composites. Les éléments simples sont constitués d'une seule action, comme l'action ASSIGN, alors que les composites sont constitués de sous-éléments comme KaBranchElement, KaWhileElement, KaWhenElement et KaDoElement.
Intention (IntentionStack)	La classe qui implante le concept d'intention. Une intention contient, entre autres, le but sélectionné par le SoakElement.
KaContext	La classe qui gère la section CONTEXT du Ka. Elle consiste en une liste de Conditions et différents services pour l'évaluer.
Structure d'intention (Intention Structure)	Elle contient les intentions sélectionnées et assure l'exécution de l'intention active. Celle-ci est déterminée en fonction de sa priorité. Une intention interrompue reprendra son exécution, si elle est toujours appropriée (après évaluation de son contexte), après l'exécution des intentions de plus hautes priorités.
Soak	Elle gère la sélection des intentions (SoakElement) à chaque cycle et offre différents services pour en choisir une.
SoakElement	La classe qui relie le but à un KA sélectionné pour le satisfaire. Les instances de cette classe ne sont créés que temporairement, le temps qu'une d'entre elle soit sélectionnée par la structure d'intention. Une intention est créée à partir de ce SoakElement.
KaRunTime-Frame	Elle gère l'exécution du KA sélectionné pour satisfaire un but. Elle est spécialisée pour chacun des KaBodyElements possibles, chacun redéfinissant son service "execute"
Action	Cette classe définit la procédure du KA. C'est l'objet de plus bas niveau pour l'exécution.
Expression	Cette classe et ses spécialisations gèrent les expressions arithmétiques et de prédicats d'UM-PRS.
Condition	Cette classe et ses spécialisations gèrent les expressions logiques utilisées dans l'UM-PRS telles que dans la commande TEST ou dans la section CONTEXT.

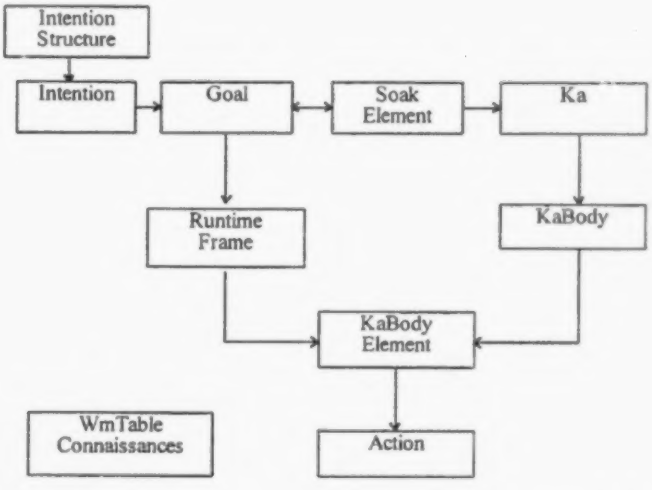


FIGURE 6 - Relations entre les principales classes d'UM-PRS

5.0 PERFORMANCES DU SYSTÈME UM-PRS

Dans ce chapitre, nous présentons les résultats de tests effectués sur le système UM-PRS afin d'évaluer:

- la performance de l'accès à la base de données;
- la dégradation du système en fonctions du nombre de buts;
- la performance de la génération de SOAK et l'exécution de KA.

Ces tests ne sont pas exhaustifs et n'ont pas la prétention de caractériser totalement la performance du système. Toutefois, les résultats obtenus ont permis de nous forger une idée assez précise des avantages et désavantages qu'offre le système UM-PRS, ce qui était le but de notre démarche.

5.1 Aspects qui influencent la performance du système

Les hypothèses que nous émettons pour évaluer la performance d'UM-PRS sont:

- la performance d'UM-PRS dépend principalement du temps d'accès aux bases de connaissances;
- le temps d'exécution de la boucle de contrôle varie surtout en fonction du temps consacré à la phase de génération de SOAK.

Les énoncés appuyant ces hypothèses figurent dans les paragraphes suivants.

5.1.1 Aspects liés aux temps d'accès à la base de données d'UM-PRS

La performance d'UM-PRS est intimement liée au temps d'accès aux bases de connaissances, puisque la plupart des opérations de la boucle de contrôle consultent la base de données et la librairie de KA (toutes deux utilisant la même structure de données).

Puisque la base de données est gérée à l'aide d'une simple table de hashing, nous nous attendons à ce que la consultation des faits dans la base de données soit sujette aux avantages et limitations suivantes:

- un temps quasi constant pour une répartition homogène des prédicats;
- un temps linéairement proportionnel au nombre d'entrées (faits) d'un même prédicat.

L'aptitude du système à détruire les faits devrait être similaire à la consultation puisqu'il faut d'abord trouver l'élément dans la base de données avant de le retirer.

Finalement, le temps d'insertion devrait, quant à lui, être toujours constant (propriété d'une table de hashing).

5.1.2 Aspects liés à la performance de la boucle de contrôle d'UM-PRS

Du côté de la boucle de contrôle, un cycle d'exécution peut se diviser en trois étapes: (i) la génération du SOAK, (ii) la sélection à partir de la structure d'intentions et (iii) l'exécution d'une intention.

Nous prévoyons que le temps de sélection à partir de la structure d'intentions (étapes ii) est négligeable, parce que l'absence de méta-KA dans le système UM-PRS réduit cette sélection à une comparaison de valeurs de priorités.

Nous présumons que le temps d'exécution d'une intention (étape iii) sera quasi constant. Rappelons que l'exécution d'une intention implique soit la génération de sous-buts, soit l'exécution d'une action primitive. Cette hypothèse est donc valide pour les KA dont les actions primitives exigent peu de resources de traitement (i.e. dont le temps d'exécution est proportionnel à celui de la génération d'un but); cette situation s'applique aux systèmes réactifs exigeant une bonne rapidité d'exécution.

Pour la génération du SOAK (étape i), nous estimons que le temps requis est proportionnel au nombre de buts dans la structure de buts, puisque la structure est entièrement consultée à chaque cycle. Pour chaque but, les KA applicables doivent être évalués en fonction de leur contexte. Or, le temps d'évaluation du KA dépend du nombre de conditions définies dans sa section CONTEXT. Le temps requis pour la génération du SOAK peut donc être exprimé par l'équation suivante:

$$TCycle = \sum_{i=0}^{n} TGoal_i$$

$$\text{où} \quad TGoal = \sum_{0}^{m} TKa_j \quad \text{et} \quad TKa = \sum_{k=0}^{c} TCondition_k \quad .$$

où *TCondition* le temps de vérification d'une condition, *n* représente le nombre de buts, *m* le nombre de KA et *c* le nombre de conditions présentes dans le contexte du KA.

Ces équations indiquent que le temps d'exécution d'un cycle augmente rapidement en fonction du nombre de buts et de KA à préconditions complexes. Le temps de vérification d'une condition est donc le facteur déterminant pour évaluer la performance de la boucle de contrôle puisque c'est cette opération qui, à chaque cycle, est exécutée le plus souvent. Comme l'illustre la fig. 7, on assiste rapidement à une explosion de conditions, et ainsi à autant de consultations sur la base de données.

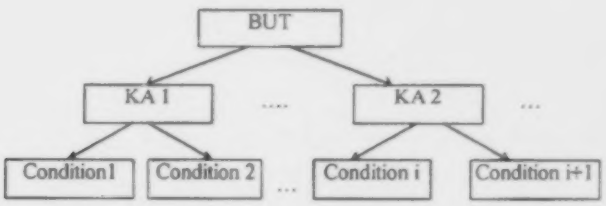


FIGURE 7 - Ramification des conditions à vérifier

5.2 Résultats des tests

Le premier test effectué consiste à déterminer le temps de création et de destruction d'un groupe de faits de même prédicat. Les résultats[1], présentés à la fig. 8, indiquent que le temps de création ne croît pas linéairement en fonction du nombre d'insertions (mais plutôt quadratiquement). Ce n'est pas le comportement auquel on s'attendait pour une simple table de hashing. Par contre, le temps de destruction est plutôt linéaire, bien que l'on note quelques variations qui sont probablement dues à la façon dont la mémoire est libérée par le système d'exploitation.

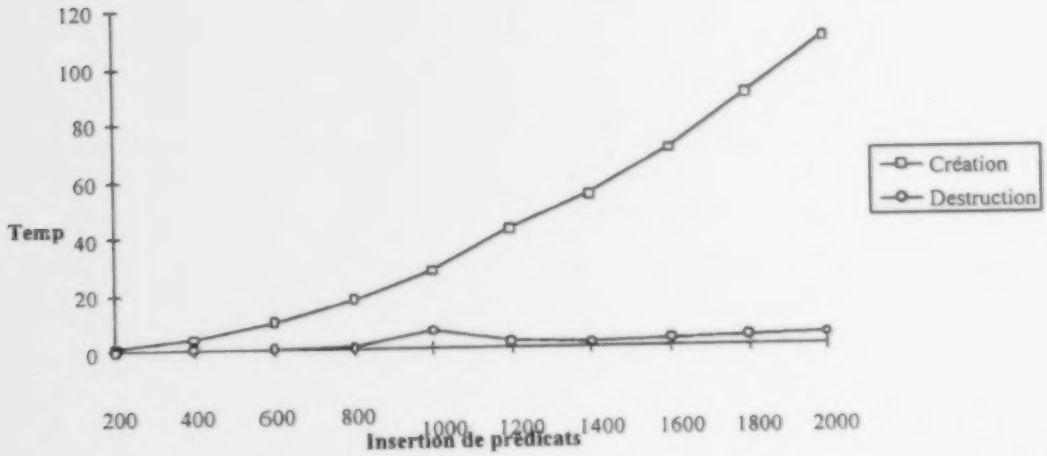


FIGURE 8 - Temps de création et de destruction de N faits de même prédicat

Le second test consiste à déterminer le temps nécessaire pour accéder 200 fois à un fait d'une base de données dont tous les faits sont du même prédicat. Comme on pouvait le prévoir, les résultats présentés à la fig. 9 indiquent que le temps d'accès est linéaire.

[1] Les résultas présentés ne comportent pas de marge d'erreur.

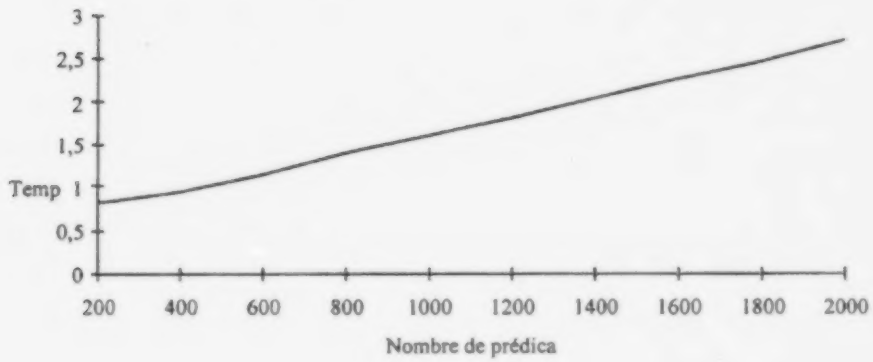


FIGURE 9 - Temps d'accès (200 répétitions)

Le troisième test consiste à déterminer le temps qu'il faut pour satisfaire un certain nombre de buts avec seulement un KA toujours applicable. Les résultats (Fig. 10) nous montrent que le temps augmente non linéairement en fonction du nombre de buts dans le système. Ce comportement était à prévoir, puisque l'on tient compte de tous les buts pour la construction du SOAK à chaque cycle.

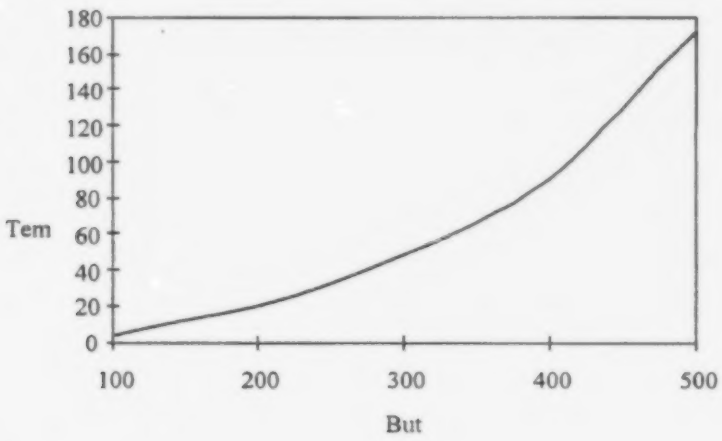


FIGURE 10 - Temps requis pour la satisfaction de N buts

Le dernier test a permis de déterminer le temps requis pour effectuer les deux actions principales de la boucle de contrôle, soit la génération du SOAK et l'exécution

d'une instruction du KA sélectionné. Les résultats (fig. 11) indiquent une fois de plus que la réactivité d'UM-PRS est directement liée au nombre de buts à satisfaire. Le temps de génération augmente non linéairement (tendance quadratique) en fonction du nombre de buts dans le système. Le temps d'exécution semble linéaire. A chaque cycle, une seule action est complétée. Si cette action est une action primitive, alors le temps d'exécution est constant. Par contre, si l'action implique la génération d'un sous-but, il est alors normal que le temps d'exécution augmente linéairement en fonction du nombre de buts; la raison étant que la structure de but est entièrement consultée afin d'éviter d'y introduire un but redondant.

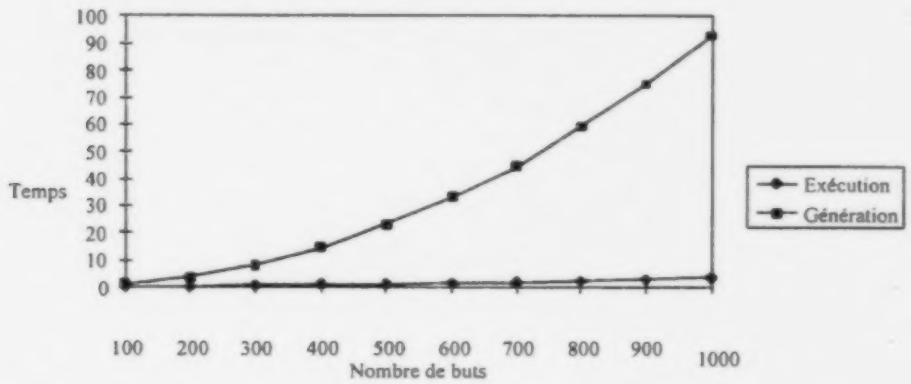


Figure 10 - Temps requis pour la génération du SOAK et l'exécution d'une action de KA

6.0 CONCLUSION

Nous avons présenté dans ce document les principes généraux des systèmes à raisonnement procédural. Nous avons répertorié les différents logiciels disponibles au moment où l'étude était menée et nous avons évalué l'un de ces logiciels [2], le système UM-PRS.

[2] Les auteurs de ce document tiennent à remercier le professeur E. Durfee, de l'Université du Michigan pour avoir rendu disponible le système UM-PRS.

D'un point de vue général, le système UM-PRS est conforme en grande partie au modèle initial des systèmes à raisonnement procédural, mais il possède quelques différences qui le rendent moins attrayant.

Les principaux avantages qu'offre ce système sont les suivants:

- il est programmé en C++ standard,
- le logiciel est bien structuré selon une modélisation orientée-objet,
- le noyau du système est facilement extensible par la définition de nouvelles primitives,
- le logiciel est disponible gratuitement.

Du coté des désavantages, on note principalement l'absence de méta-KA et de mécanisme de déclenchement à partir de nouvelles données. De plus, un environnement de conception adéquat est nécessaire avant d'entreprendre le développement d'une application avec ce logiciel.

L'objectif de notre étude était d'évaluer le potentiel de ce système comme infrastructure pour un système adaptatif intelligent. Nous avons décidé de ne pas opter pour le système UM-PRS pour les raisons suivantes:

- la représentation du domaine ("World Model") d'UM-PRS, de type logique du premier ordre, ne permet pas de déclarer de faits contenant des variables. Il devient alors préférable d'adopter une représentation du domaine de type orienté-objet.
- les résultats du premier test, indiquant la non-linéarité de la création de nouveaux faits, mettent en doute la qualité de la mise en oeuvre de l'accès à la base de données;
- l'absence d'une couche de méta-niveau (méta-KA ou autre concept similaire) est une sérieuse lacune puisqu'elle aurait pu être utilisée pour délibérer dynamiquement sur le cours des actions executées par le système.

7.0 RÉFÉRENCES

[1] Georgeff, M.; Lansky, A.; "Procedural Knowledge", Proceedings of IEEE Special issue on Knowledge Representation, 74:1283-1398, 1986.

[2] Georgeff, M. P.; Ingrand, F. F.;. "Decision-Making in an Embedded Reasoning System", Proceedings of IJCAI-89, pp. 972-978, 1989.

[3] Wilkins, D. E.; Myers, K. L.; Lowrance, J. D.; Wesley, L. P.; "Planning and Reacting in Uncertain and Dynamic Environments"; Artificial Intelligence Center, SRI International; Journal of Experimental and Theoretical AI, volume 6, 1994, pp. 197-227.

[4] Ingrand, F. F.; Coutance V.; "Procedural Reasoning versus Blackboard Architecture for Real-Time Reasoning", Proceeding of the Thirteenth International Conference on Artificial Intelligence, Avignon, 1993.

[5] Ingrand, F. F.; Chatila, R.; Alami, R.; Robert, F.; "PRS: A High Level Supervision and Control Language for Autonomous Mobile Robots", Proceeding of the 1996 IEEE International Conference on Robotics and Automation . Minneapolis, USA.

[6] Huber, M. J.; Lee, J.; Kenny, P.; Durfee, E. H.; "UM-PRS V3.0, Programmer and User Guide", The University of Michigan, February 15, 1996.

[7] Lee, J.; Huber, M. J.; Durfee, E. H.; Kenny, P. G; "UM-PRS: An Implementation of the Procedural Reasoning System for Multirobot Applications", Proceedings of the 1990 DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control, pages 284-291, San Diego, CA, Novembre 1990.

ANNEXE A
Description des fichiers du système UM-PRS

Fichiers	Descriptions
action.cc, action.h	Implantation de la classe Action ainsi que ses spécialisations
binding.cc, binding.h	Classes BindingValue et Binding. Gère le binding d'un but avec un KA
condition.cc, condition.h	Implantation des classes de conditions logiques: Condition, ExpCondition, RelCondition, FactCondition, RetrieveCondition
dlist.h	Implantation d'un "template" de liste doublement chaîné
exp-fun.h	Implantation de la classe Funcall qui gère l'appel de primitives
exp-val.cc, exp-val.h	Classes qui gèrent les expressions logiques et arithmétiques
exp-var.h	Implantation de la classe Variable
exp.h	Implantation de la classe Expression et de ses spécialisations
extern.cc, extern.h	Implantation la classe Program et Parser
goal.cc, goal.h	Implantation de la classe Goal et GoalList
is.cc, is.h	Implantation de la classe IntentionStructure
ka.cc, ka.h	KaBodyDoElement, KaBodyElement, KaBodySimpleElement, KaBodyWhileElement
lex.yy.c	Analyseur lexical généré par flex avec le fichier yytoken.l
match.cc, match.h	Implantation des fonctions d'inférence ("pattern matching")
parse.cc, parse.h	Implantation des fonctions pour le *parsing* des scripts
predicate.h	Classe Predicate et ses spécialisation: PredicateAchieve, PredicateFact et PredicateRetrieve
primitive.cc, primitive.h	Implantation des primitives du système et des macros pour la définition de primitives par l'usager
random.h	Implantation des classes Random_Exponential, Random_Uniform_Double et Random_Uniform_Integer. Utilisé dans la sélection d'un SoakElement
relation.cc, relation.h	Implantation de la classe Relation
run.cc, run.h	Implantation des classes de RunTimeFrame et ses spécialisations
slist.h	Implantation du "template" Slist, une liste simplement chaîne
soak.cc, soak.h	Implantation des classes Soak, SoakElement et SoakList
symtab.cc, symtab.h	Implantation des classes SymbolRep, Symbol et SymbolTable
um-prs.cc, um-prs.h	Implantation de la boucle de contrôle d'UM-PRS
user.h	Fichier regroupant les "headers" à inclure pour utiliser la librairie um-prs.a
wm.cc, wm.h	Implantation des classes WmRelation, WmTable, des spécialisations de Symbol et SymbolTable pour la gestion du WorldModel
yyparse.c, yyparse.h	Fichiers générés par *bison*, un générateur de *"parser"*, en utilisant yyparse.y Implantation du *"parser"* qui fait la lecture des fichiers de commandes

ANNEXE A
Description des fichiers du système UM-PRS

Fichiers	Descriptions
action.cc, action.h	Implantation de la classe Action ainsi que ses spécialisations
binding.cc, binding.h	Classes BindingValue et Binding. Gère le binding d'un but avec un KA
condition.cc, condition.h	Implantation des classes de conditions logiques: Condition, ExpCondition, RelCondition, FactCondition, RetrieveCondition
dlist.h	Implantation d'un "template" de liste doublement chaîné
exp-fun.h	Implantation de la classe Funcall qui gère l'appel de primitives
exp-val.cc, exp-val.h	Classes qui gèrent les expressions logiques et arithmétiques
exp-var.h	Implantation de la classe Variable
exp.h	Implantation de la classe Expression et de ses spécialisations
extern.cc, extern.h	Implantation la classe Program et Parser
goal.cc, goal.h	Implantation de la classe Goal et GoalList
is.cc, is.h	Implantation de la classe IntentionStructure
ka.cc, ka.h	KaBodyDoElement, KaBodyElement, KaBodySimpleElement, KaBodyWhileElement
lex.yy.c	Analyseur lexical généré par flex avec le fichier yytoken.l
match.cc, match.h	Implantation des fonctions d'inférence ("pattern matching")
parse.cc, parse.h	Implantation des fonctions pour le *parsing* des scripts
predicate.h	Classe Predicate et ses spécialisation: PredicateAchieve, PredicateFact et PredicateRetrieve
primitive.cc, primitive.h	Implantation des primitives du système et des macros pour la définition de primitives par l'usager
random.h	Implantation des classes Random_Exponential, Random_Uniform_Double et Random_Uniform_Integer. Utilisé dans la sélection d'un SoakElement
relation.cc, relation.h	Implantation de la classe Relation
run.cc, run.h	Implantation des classes de RunTimeFrame et ses spécialisations
slist.h	Implantation du "template" Slist, une liste simplement chaîne
soak.cc, soak.h	Implantation des classes Soak, SoakElement et SoakList
symtab.cc, symtab.h	Implantation des classes SymbolRep, Symbol et SymbolTable
um-prs.cc, um-prs.h	Implantation de la boucle de contrôle d'UM-PRS
user.h	Fichier regroupant les "headers" à inclure pour utiliser la librairie um-prs.a
wm.cc, wm.h	Implantation des classes WmRelation, WmTable, des spécialisations de Symbol et SymbolTable pour la gestion du WorldModel
yyparse.c, yyparse.h	Fichiers générés par *bison*, un générateur de *"parser"*, en utilisant yyparse.y. Implantation du *"parser"* qui fait la lecture des fichiers de commandes

ANNEXE B

Description des classes du système UM-PRS

B.1 Conventions utilisées

Après une brève description, les classes sont décrites de la façon suivante:

Attributs

nom_attribut :Type *Description*

...

Services

nom_service(arguments)
Description

...

B.2 Program

La classe Program contient les principaux éléments d'UM-PRS ainsi que les options sélectionnées lors de son exécution. Une instance de la classe Program, *program*, est créée lors de l'exécution et est visible globalement par toutes les autres composantes.

Attributs

ka_library :KaTable	*Liste des KA*
world_model :WmTable	*Base de connaissances*
goal_list :GoalSet	*Objet qui gère la structure des buts*
cycle :Ka	*KA à exécuter à tous les cycles*
show_wm :int	*Option pour afficher le WorldModel lorsqu'il est modifié*
show_gl :int	*Option pour afficher la structure de buts lorsqu'elle est modifiée*
show_is :int	*Option pour afficher la structure d'intention lorsqu'elle est modifiée*
show_soak :int	*Option pour afficher le SOAK à sa création.*
sim_mode :int	*Option pour exécution en mode simulation.*
priority_evaluation_function_hook	*Fonction que l'utilisateur peut donner pour l'évaluation de la priorité. Cette fonction reçoit en*

paramètres la priorité du But et celle du Ka sous forme de deux nombres réels

Services : Aucun

B.3 Relation

La classe Relation est un élément de connaissance. Elle associe un fait avec une liste de ses valeurs.

Attributs

relations :static SymbolTable	*Pointeur vers le WorldModel, la base de connaissances*
id :long	*Identifiant du fait dans le WorldModel*
args :ExpList	*Liste d'expressions*
arity :long	*Nombre d'expressions*

Services

eval_args(Binding)
Évalue les arguments après avoir fait l'instanciation des variables avec le Binding

Comme les arguments de la relation sont des expressions, cela permet l'insertion de primitives dans la base de connaissances. Celles-ci seront évaluées lors de la manipulation de la connaissance.

B.4 WmTable (WorldModel)

La classe WmTable est une spécialisation de SymbolTable, qui est une table de hashing. Cette classe gère la base de connaissances et est composée de Relations. Une relation peut être retrouvée par son nom ou, pour être plus performant, par son ID en exploitant la fonction de hashing. L'ID est le plus souvent utilisé par les autres classes afin de conserver un lien avec une Relation.

Attributs: *Aucun.*

Services

assert(Relation, Binding)
Ajoute une Relation, en l'instanciant d'abord avec le Binding

lookup(id)
Retrouve une Relation d'après son ID

lookup(name)
Retrouve une Relation d'après son nom

retract(Relation, Binding)
Enlève une relation, en l'instanciant d'abord avec le Binding

replace(Relation, Relation)
Remplace une relation par une autre

B.5 Goal

La classe Goal décrit la notion de but. Des instances de cette classe sont insérées dans la structure des buts lors de l'initialisation du système, dynamiquement avec la commande POST ou en exécutant un sous-but.

Attributs

goal :GoalAction	*Pointeur vers l'action qui à généré ce but (ex:. ACHIEVE Goal1 "Robot1")*
subgoal :Goal	*Pointeur vers un sous-but si un est créé en cours de réalisation de celui-ci*
prev_goal :Goal	*Si le but est un sous-but ceci pointe vers celui qui l'a créé*
new_goal :int	*Détermine si un but est nouveau ou non*
status: StatusType	*IS_ACTIVE, IS_SUSPENDED, IS_SUCCESS, IS_FAILURE, IS_UNTRIED, IS_ABANDONED*
Intention (SoakElement)	*Pointeur vers l'intention qui tente d'atteindre ce but*
RuntimeFrame	*Objet qui pointe vers l'instruction du Ka à exécuter*
intention_structure: IntentionStructure	*Pointeur vers la structure d'intention dans laquelle le but à été sélectionné*

Services

eval_priority()
Évalue la priorité du but

generate_soak()
Détermine si un SOAK doit être généré pour ce but, c'est-à-dire s'il n'est pas déjà sélectionné ou s'il n'a pas déjà été atteint

get_current_goal()
Donne le But qui est activé par la structure d'intention

is_leaf_goal()
Détermine si un but est le dernier dans la hiérarchie, i.e. qu'il n'a pas de sous-buts

is_top_level_goal()
Détermine si un but est un but principal, c'est-à-dire qu'il n'a pas de but parent

remove_intention(failed)
Enlève les sous-buts de la structure des buts, met l'attribut d'intention à 0 et exécute la section failure du Ka si "failed=vrai"

B.6 GoalSet

La classe GoalSet gère la structure des buts et offre des services pour la manipuler.

Attributs

goals :DList<Goal> *Liste doublement chaînée de buts*

Services

add(Goal)
Ajoute un but à la structure (de buts)

all_goal_done()
Détermine si tous les buts de la liste sont satisfaits

drop(GoalAction, Binding)
Enlève le but de la structure des buts. Son statut est modifié pour IS_ABANDONNED. Il ne sera donc plus retenu

remove(Goal)
Enlève le but de la liste pour évaluation immédiate, mais demeure accessible pour résolution future.

B.7 Ka

La classe KA décrit la notion de KA telle que décrite précédemment Ses services portent surtout sur la manipulation des conditions d'activation. L'exécution du KA est gérée par la structure d'intention, les services du KA à ce sujet se limitant à créer un "run-time frame" approprié pour l'action en cours d'exécution.

Attributs

*name: char *.*	*Chaîne de caractères décrivant le KA (NAME:)*
*documentation: char *.*	*(DOCUMENTATION:)*
purpose: action.	*Une action de type ACHIEVE (PURPOSE:)*
contexte: KaContext	*Liste de conditions. Lors du test de sélection d'un KA, chaque des conditions est évaluée et le KA est rejeté à la première qui échoue (CONTEXT:)*
priority: Expression	*Expression qui décrit la priorité de ce KA. (PRIORITY:).*
body: KaBody	*Objet qui pointe vers le premier KaBodyElement. Le corps est constitué d'une liste chaînée de KaBodyElement. Chaque KaBodyElement correspond à une commande donnée dans le script. Lors de l'exécution, les KaBodyElement sont traversés et exécutés par un KaRuntimeFrame (BODY:).*
effect: KaBody	*Objet qui pointe vers le premier KaBodyElement (EFFECT:)*
failure: KaBodyAtomicElement	*Action à exécuter si le KA échoue en cours d'exécution (FAILURE:)*

valide: int	*Détermine si le KA est valide ou non. Seul les KA valides seront retenus lors de la génération du SOAK. Le concepteur peut donc désactiver un KA à l'aide de cet attribut.*

Services

eval_priority(Binding)
Évalue la priorité du KA en fonction du Binding

check_context(BindingList)
Enlève les Bindings qui ne répondent pas aux conditions du contexte

confirm_context(Binding)
Détermine si un Binding satisfait aux conditions du contexte

disable()
L'attribut valid du KA est mis à 0

is_valid()
Détermine si le KA est valide (valid=1)

B.8 KaBodyElement

Le KaBodyElement est l'élément de base dont est constitué le corps d'un KA. Les KaBodyElement sont groupés en listes chaînées constituant les actions à exécuter. On peut classer les KaBodyElements en deux groupes: les éléments simples et les éléments composites. Les éléments simples sont constitués d'une seule action, comme ASSIGN, alors que les composites sont constituées de sous-éléments comme KaBranchElement, KaWhileElement, KaWhenElement et KaDoElement.

Attributs

next_element :KaBodyElement	*Pointeur vers le prochain KaBodyElement*
element_type :KaElementType	*Type de l'élément: KA_SIMPLE, KA_BRANCH, KA_WHILE, KA_WHEN, KA_ATOMIC ou KA_DO*

Services

new_run_time_frame()
Crée un KaRuntimeFrame approprié pour le type d'élément

get_next_element()
Donne l'élément suivant dans la liste

is_final_element()
Détermine s'il s'agit du dernier élément

Spécialisations

KaBodySimpleElement
Gère les actions simples comme ASSIGN, TEST, etc.

KaBodyDoElement
Gère la commande DO

KaBodyBranchElement
Gère les instructions des branchements AND et OR

KaBodyAtomicElement.
Gère la commande ATOMIC

KaBodyWhenElement
Gère la commande WHEN

KaBodyWhileElement
Gère la commande WHILE

Le KaBodyWhileElement est composé d'une "Action Test" ainsi que d'un pointeur vers son premier KaBodyElement

Lors de l'exécution du KaBodyWhileElement, le test est effectué au tout début. Si le résultat est positif, la liste des commandes est exécutée. L'exécution ne se termine que lorsque "Action test" retourne un résultat négatif.

B.9 Intention (IntentionStack)

IntentionStack est la classe qui implante le concept d'intention. Une intention contient principalement le but sélectionné par le SoakElement.

Attributs

top_level_goal : Goal	*But de niveau principal*
current_goal : Goal	*But sélectionné (peut être un sous-but)*

Services

get_current_goal(): Goal
Retourne le but courant

get_top_level_goal(): Goal
Retourne le but principal d'où provient le but courant

set_current_goal(Goal)
Remplace le but courant par celui passé à la fonction

set_top_level_goal(Goal)
Remplace le but principal par celui passé à la fonction.

B.10 KaContext

KaContext est la classe qui gère la section CONTEXT du KA. Elle consiste en une liste de conditions et différents services pour l'évaluer.

Attributs

conditions :ConditionList liste de conditions

Services

check(BindingList) :int
Enlève les bindings de la liste qui ne satisfont pas aux conditions

confirm(Binding) :int
Détermine si le Binding satisfait aux conditions

B.11 Structure d`Intention (IntentionStructure)

La structure d'intention contient les intentions sélectionnées et assure l'exécution de l'intention active. Celle-ci est déterminée en fonction de leur priorité. L'exécution d'une intention interrompue reprendra, si elle est toujours appropriée (après évaluation de son contexte), après l'exécution des intentions de priorités supérieurs.

Attributs

top_level_goals : IntentionStackListListe d'intentions

current_stack : IntentionStack Intention active

Services: *Il y a deux services principaux pour IntentionStructure: activate qui tente d'exécuter une instruction d'un Ka sélectionné et intend qui accepte un SoakElement comme argument et détermine si celui-ci sera immédiatement exécuté ou mis en attente dans la liste*

activate

1. S'il n'y a pas d'intention active, le résultat est IS_SUCCESS.

2. Le but de niveau supérieur et son RuntimeFrame sont sélectionnés à l'aide des variables temporaires top_goal: Goal et top_frame: RuntimeFrame.

top_goal = current_stack.top_level_goal

top_frame = top_goal.runtime_frame

3. Le contexte du Ka est validé ainsi que le statut du but.

Si le contexte n'est pas valide ou si le but à le statut IS_ABANDONNED le but et ses sous-buts sont enlevés de la structure de but et une autre intention est sélectionnée.

4. Le KA est exécuté par l'entremise du top_frame.execute(goal.intention.Binding, current_goal) et peut avoir trois résultats possibles.

4.1. Si le résultat est KA_ELEMENT_FAILURE

Si une section FAILURE existe, elle est exécutée.

L'intention du but est enlevé de la structure d'intention, les sous-buts sont enlevés du GoalList. L'intention de plus haute priorité est ensuite sélectionnée par la structure d'intention.

Le résultat retourné est IS_FAILURE.

4.2. Si le résultat est KA_ELEMENT_INCOMPLETE

Le résultat retourné est IS_ACTIVE.

4.3. Si le résultat est KA_ELEMENT_COMPLETE

S'il s'agit de la dernière action, le statut du but est changé pour IS_SUCCESS, il est enlevé de la structure de but et une autre intention est sélectionnée dans la structure d'intention.

S'il ne s'agit pas de la dernière action, la prochaine action est sélectionnée dans le runtime_frame.

Le résultat retourné est IS_ACTIVE.

intend(SoakElement element): SoakElement

1. La structure d'intention est balayée à la recherche d'une intention existante pour le but principal de ce SoakElement.

current_stack = find_goal_in_stack(element.from_goal)

2. S'il n'y a pas d'intention dans la liste, une intention est créée pour ce but. Celle-ci est sélectionnée par la structure d'intention.

3. Sinon, s'il s'agit d'un but principal ou si ce but existe dans la liste, sa priorité détermine s'il remplacera celui déjà sélectionné.

3.1. Si sa priorité est supérieur à tous ceux de la liste ou si ceux de priorité supérieure ont IS_BLOCKED comme statut, l'intention est sélectionnée.

3.2. Sinon, son statut devient IS_ACTIVE et l'intention est insérée dans la liste.

ou si le statut du but sélectionné est IS_BLOCKED il remplace l'ancien.

4. Si c'est un sous-but, le but courant de l'intention est remplacé par le sous-but.

5 . Le runtime_frame est créé en prévision de la prochaine exécution.

clear_current_stack
Enlève l'intention sélectionnée de la structure d'intention

execute_cycle : StatusType
Exécute la section CYCLE

find_goal_in_stacks(Goal) : IntentionStack
Trouve l'IntentionStack qui contient Goal

find_toplevel_goal(Goal) : IntentionStack
Trouve l'IntentionStack qui contient le but principal parent de Goal

get_current_goal : Goal
Retourne le but qui est sélectionné pour exécution

get_top_level_goal : Goal
Retourne le but principal associé au but sélectionné pour exécution

refresh_stack_priorities
Recalcule la priorité des intentions

B.12 Soak

La classe Soak gère la sélection des intentions (SoakElement) à chaque cycle et offre différents services pour en choisir une.

Attributs

intentions	Liste de SoakElements

Services

get_first() :SoakElement
Donne le premier SoakElement

get_priority_first() :SoakElement
Donne le premier SoakElement en considérant les priorités

get_priority_random() :SoakElement
Donne un SoakElement au hasard parmi celles de priorités supérieures

instanciate(Ka, Goal, WmTable)
Crée un SoakElement et l'ajoute à la liste si les conditions du PURPOSE, CONTEXT réussissent

get_random() :SoakElement
Retourne un SoakElement sélectionné au hasard

Génération du SOAK: La génération du SOAK est effectuée à chaque itération de la boucle de contrôle du système UM-PRS (voir *Boucle de contrôle*, page 6).

Étapes de la génération:

- *Chaque but est évalué pour déterminer s'il doit être inclus dans le SOAK. Le but ne doit pas être déjà atteint et ne doit pas posséder de "sous-buts" actifs (ce qui indiquerait que le but est déjà sélectionné.)*
- *Pour chaque but répondant à ces critères, l'ensemble des KA est balayé pour trouver ceux qui s'appliquent.*
- *Pour chaque KA, un "pattern matching" est effectué entre la section PURPOSE du KA et celle du but.*
- *Si le "pattern matching" réussit, un Binding est créé sinon le KA est rejeté.*
- *La section CONTEXTE du KA est ensuite vérifiée avec le Binding.*
- *Si toutes ces étapes réussissent, un SoakElement est créé avec comme paramètres le But, le KA et le Binding. Le SoakElement est alors ajouté au SOAK.*
- *À la suite de cette génération, le SoakElement de priorité supérieur est sélectionné et présenté à la structure d'intention avec son service intend. Une intention (IntentionStack) sera créée à partir de ce SoakElement.*

B.13 SoakElement

Le SoakElement est l'objet qui relie le but avec un KA sélectionné comme pouvant l'atteindre. Les SoakElements ne sont créés que temporairement, le temps que l'un d'eux ne soit sélectionné par la structure d'intention. Une intention est créée à partir de ce SoakElement.

Attributs

ka: Ka	*KA sélectionné par le Soak*
but: Goal	*But sélectionné par le Soak*
binding: Binding	

Services

eval_priority()
Donne la somme de la priorité du but et celle du KA ou appelle une fonction d'évaluation de priorité fournie par l'utilisateur. Cette fonction reçoit en paramètres les deux priorités sous forme de nombres réels.

B.14 KaRunTimeFrame

La classe KaRunTimeFrame gère l'exécution du KA sélectionné pour atteindre un but. Cette classe est spécialisée pour chacun des KaBodyElements, chacun redéfinissant son service "execute"

Attributs

this_element	*L'élément qui a créé le RunTimeFrame*
sub_frame	*Un sous RuntimeFrame utilisé lors de l'exécution d'actions composite.*

Services

execute (Binding, Goal)
Exécute l'action associée dans le cas d'élément simple ou génère le sub_frame dans le cas d'action composite

Lors d'une action composite le service execute se fait en cascade en appelant l'execute du sub_frame. Tant que la dernière action n'est pas atteinte ce dernier retourne KA_INCOMPLETE.

Lors de l'exécution d'un runtime_frame, la partie sub_frame est d'abord retenue. Si celle-ci existe, son service execute est appelé jusqu'à ce que celui-ci a pour résultat KA_COMPLETE ou KA_FAILURE. Sur réception de l'un de ces derniers, l'exécution de la trame principale peut être complétée. C'est de cette façon que le système garde la trace de son exécution. Dans le cas d'un sous-but, la spécialisation KaGoalRuntimeFrame est utilisée. Celui-ci possède en plus un attribut Goal qui possède à son tour son propre runtime_frame.

B.15 KaGoalRuntimeFrame

Attributs

subgoal :Goal

Services

execute

Le service execute du KaGoalRuntimeFrame se comporte essentiellement de la même façon que le service activate de la structure d'intention lors de l'exécution du runtime_frame. Au lieu d'utiliser l'attribut subframe, le runtime_frame du but sera utilisé pour l'exécution.

B.16 Action

La classe Action est l'objet qui accomplit la procédure du KA. C'est l'objet de plus bas niveau pour l'exécution.

Attributs

un nom

nom du fichier source

ligne du fichier source qui a généré cette action

Services

execute(Binding)
Exécute l'action. Ce service est redéfini dans toutes les spécialisations

Spécialisations

La classe Action se spécialise pour tous les types de commandes disponibles dans le langage. Son service execute est redéfini pour chacun d'eux.

Assign, Fact, Fail, GoalAction (Achieve, Maintain, Query, Wait), Load, Parse, Post, Primitive, Retract, Retrieve, Test, Unpost et Update.

B.17 Expression

La classe Expression et ses spécialisations gèrent les expressions arithmétiques et de prédicats du système UM-PRS.

Services

check(BindingList) :int
Enlève les bindings de la liste qui ne satisfont pas aux conditions

confirm(Binding) :int
Détermine si le Binding satisfait aux conditions

Eval(Binding) :Value
Evalue une expression et retourne le résultat

B.18 Condition

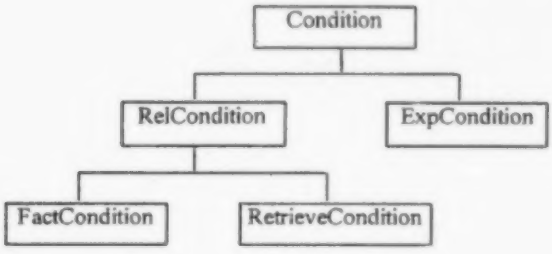
La classe Condition et ses spécialisations gèrent les expressions logiques utilisées dans le système UM-PRS, telles que la commande TEST ou la section CONTEXT:

Services

check(BindingList) :int
Enlève les bindings de la liste qui ne satisfont pas aux conditions

confirm(Binding) :int
Détermine si le Binding satisfait aux conditions

41


Condition
RelCondition
ExpCondition
FactCondition
RetrieveCondition

ANNEXE C

Solutions proposées pour les évènements déclencheurs

C.1 Première solution

L'une des solutions identifiées, à la suite des discussions que nous avons eues avec les concepteurs d'UM-PRS, consiste à insérer, dans la structure des buts, tous ceux qui sont susceptibles de se présenter dans le système. Le contexte du KA assurera que ce but ne soit sélectionné que lorsque la modification à la base de données permettra l'activation de ce but. Bien qu'elle fonctionne, cette solution demeure peu pratique.

Dans l'exemple suivant, le but à atteindre est *ReachPointC.* On débute par l'ajout de plans au système par l'entremise du KA *InitPlans.* On tente ensuite d'atteindre le but *ReachPointB.* Le KA qui s'applique à *ReachPointB* est en une boucle où la primitive *move* est appelée. La création d'obstacles aléatoires qui seront insérés dans la base de données est déterminée dans cette dernière. Lorsqu'on introduit un obstacle, le système réagit immédiatement (en raison de sa priorité élevée) en faisant appel au KA *MoveAroundObstacle* ou *MoveOverObstacle* selon la hauteur de l'obstacle. Les plans sont finalement enlevés (par le KA *RemovePlans*) de la structure de buts lorsque le but est atteint.

```
GOALS:
        ACHIEVE ReachPointC "Main" :PRIORITY 10;

FACTS:
        obstacleCount 0;

//-------------------------------------
// ReachPointC
//-------------------------------------
KA (
NAME:
        "ReachPointC"
DOCUMENTATION:
        "None."
PURPOSE:
        ACHIEVE ReachPointC;
BODY:
        ACHIEVE InitPlans;
        ACHIEVE ReachPointB;

        EXECUTE print "PointC Reached\n";
```

```
        ACHIEVE RemovePlans;
        RETRIEVE obstacleCount $aCount;
        EXECUTE print "Obstacles rencontres: " $aCount "\n";

FAILURE:
        EXECUTE print "PointC failed\n";
}

//-------------------------------------
// MoveAroundObstacle
//-------------------------------------
KA {
NAME:
        "MoveAroundObstacle"
DOCUMENTATION:
        "None."
PURPOSE:
        ACHIEVE ReactToObstacle;
CONTEXT:
        FACT Obstacle $x $y $height;
        (< $height 5);
BODY:
        RETRIEVE obstacleCount $count;
        UPDATE (obstacleCount) (obstacleCount (+ $count 1));
        EXECUTE print "MoveAroundObstacle " $x " " $y  " Achieved\n";
        POST ACHIEVE ReactToObstacle :PRIORITY 100;
        RETRACT Obstacle $x $y;
FAILURE:
        EXECUTE print "MoveAroundObstacle failed.\n";
}

//-------------------------------------
// MoveOverObstacle
//-------------------------------------
KA {
NAME:
        "MoveOverObstacle"
DOCUMENTATION:
        "None."
PURPOSE:
        ACHIEVE ReactToObstacle;
CONTEXT:
        FACT Obstacle $x $y $height;
        (>= $height 5);
BODY:
        RETRIEVE obstacleCount $count;
        UPDATE (obstacleCount) (obstacleCount (+ $count 1));
        EXECUTE print "MoveOverObstacle " $x " " $y " " $height  " Achieved\n";
        POST ACHIEVE ReactToObstacle :PRIORITY 100;
        RETRACT Obstacle $x $y $height;
FAILURE:
        EXECUTE print "MoveOverObstacle failed.\n";
}

//-------------------------------------
// ReachPointB
//-------------------------------------

KA {
NAME:
        "ReachPointB."
DOCUMENTATION:
        ""
PURPOSE:
        ACHIEVE ReachPointB $TEXT;
CONTEXT:
        FACT test_done $t;
BODY:
        ASSIGN $count 100;
        WHILE: TEST (> $count 0)
                {
                        EXECUTE print "move: " $count "\n";
                        EXECUTE move;
                        ASSIGN $count (- $count 1);
```

```
                );
        EXECUTE print "PointB Reached\n";
FAILURE:
        EXECUTE print "PointB failed.\n";
}

//-------------------------------------
// InitPlans
//-------------------------------------
KA {
NAME:
        "Init Plans"
DOCUMENTATION:
        ""
PURPOSE:
        ACHIEVE InitPlans;
CONTEXT:
BODY:
        POST ACHIEVE ReactToObstacle :PRIORITY 100;
        EXECUTE print "Plans initialized\n";
FAILURE:
        EXECUTE print "Init Plans failed.\n";
}

//-------------------------------------
// RemovePlans
//-------------------------------------
KA {
NAME:
        "Remove Plans"
DOCUMENTATION:
        ""
PURPOSE:
        ACHIEVE RemovePlans;
CONTEXT:
BODY:
        UNPOST ACHIEVE ReactToObstacle :PRIORITY 100;
        EXECUTE print "Plans removed\n";
FAILURE:
        EXECUTE print "Remove Plans failed.\n";
}
```

C.2 Deuxième solution

Une autre solution proposée consiste à ajouter une liste dans la classe Program qui contiendrait des associations de contextes avec un ou des buts. Une primitive vérifierait cette liste à chaque cycle et ajouterait dans la structure de buts ceux associés aux contextes qui sont applicables. Cette modification serait relativement mineure et assurerait une certaine compatibilité avec les versions à venir d'UM-PRS. Une modification devrait être apportée à la grammaire du langage pour reconnaître la nouvelle association contexte/but.

L'ajout au langage pourrait ressembler à ce qui suit:

```
EVENT {                                    // Nouvelle section similaire au KA
        NAME:
                "Event";
        CONTEXT:
                FACT Tempete;
                RETRIEVE temperature $t;
                (< $t -10);
        GOALS:                             // Liste de buts à trigger
                ACHIEVE AppliquerSable;
                ACHIEVE Déneiger;
    }
```

L'exemple précédent pourrait être simplifié comme suit:

```
GOALS:
        ACHIEVE ReachPointC "Main" :PRIORITY 10;

FACTS:
        obstacleCount 0;

//-------------------------------------
// ReactToObstacleEvent
//-------------------------------------
EVENT {
                NAME:
                        "ReactToObstacleEvent";
                CONTEXT:
                        FACT Obstacle $x $y $height;
                GOALS:
                        ACHIEVE ReactToObstacle;
}

//-------------------------------------
// ReachPointC
//-------------------------------------
KA {
NAME:
        "ReachPointC"
DOCUMENTATION:
        "None."
PURPOSE:
        ACHIEVE ReachPointC;
BODY:
        ACHIEVE ReachPointB;
        EXECUTE print "PointC Reached\n";
        RETRIEVE obstacleCount $aCount;
        EXECUTE print "Obstacles rencontres: " $aCount "\n";
FAILURE:
        EXECUTE print "PointC failed\n";
}

//-------------------------------------
// MoveAroundObstacle
//-------------------------------------
KA {
NAME:
        "MoveAroundObstacle"
DOCUMENTATION:
        "None."
PURPOSE:
        ACHIEVE ReactToObstacle;
CONTEXT:
        FACT Obstacle $x $y $height;
        (< $height 5);
BODY:
        RETRIEVE obstacleCount $count;
        UPDATE (obstacleCount) (obstacleCount (+ $count 1));
        EXECUTE print "MoveAroundObstacle " $x " " $y  " Achieved\n";
```

```
        RETRACT Obstacle $x $y;
FAILURE:
        EXECUTE print "MoveAroundObstacle failed.\n";
}

//-------------------------------------
// MoveOverObstacle
//-------------------------------------
KA {
NAME:
        "MoveOverObstacle"
DOCUMENTATION:
        "None."
PURPOSE:
        ACHIEVE ReactToObstacle;
CONTEXT:
        FACT Obstacle $x $y $height;
        (>= $height 5);
BODY:
        RETRIEVE obstacleCount $count;
        UPDATE (obstacleCount) (obstacleCount (+ $count 1));
        EXECUTE print "MoveOverObstacle " $x " " $y " " $height  " Achieved\n";
        RETRACT Obstacle $x $y $height;
FAILURE:
        EXECUTE print "MoveOverObstacle failed.\n";
}

//-------------------------------------
// ReachPointB
//-------------------------------------
KA {
NAME:
        "ReachPointB."
DOCUMENTATION:
        ""
PURPOSE:
        ACHIEVE ReachPointB $TEXT;
CONTEXT:
        FACT test_done $t;
BODY:
        ASSIGN $count 100;
        WHILE: TEST (> $count 0)
                {
                        EXECUTE print "move: " $count "\n";
                        EXECUTE move;
                        ASSIGN $count (- $count 1);
                };
        EXECUTE print "PointB Reached\n";
FAILURE:
        EXECUTE print "PointB failed.\n";
}
```

La différence avec l'exemple précédent est que la gestion des buts dynamiquement créés (avec la commande POST) pour réagir aux événements est accomplie par la nouvelle commande EVENT.

SANS CLASSIFICATION

DISTRIBUTION INTERNE

CRDV-TM-9738

1 - Directeur général adjoint

1 - Scientifique en chef

6 - Bibliothèque des documents

1 - M. Luc Lamontagne (auteur)

1 - Mme Micheline Bélanger (auteur)

1 - M. Jean Berger (auteur)

1 - CS/TAD

1 - CS/TSI

1 - Dr É. Bossé

1 - Dr I. Abi-Zeid

1 - M. Stéphane Paradis

SANS CLASSIFICATION

DISTRIBUTION EXTERNE

CRDV-TM 9738

1 - DRDGI

1 - DRDGI (exemplaire non relié)

1 - DRDD

SANS CLASSIFICATION

COTE DE SÉCURITÉ DE LA FORMULE

(plus haut niveau du titre, du résumé ou des mots-clefs)

FICHE DE CONTRÔLE DU DOCUMENT

1. PROVENANCE (le nom et l'adresse)

CRDV
2459 boul. Pie XI nord
Val-Bélair, Qc
G3J 1X5

2. COTE DE SÉCURITÉ (y compris les notices d'avertissement s'il y a lieu)

SANS CLASSIFICATION

3. TITRE (Indiquer la cote de sécurité au moyen de l'abréviation (S,C, R ou NC) mise entre parenthèses, immédiatement après le titre.)

"Evaluation d'un système à raisonnement procédural (UM-PRS)" (U)

4. AUTEURS (Nom de famille, prénom et initiales. Indiquer les grades militaires, ex.: Bleau, Maj. Louis E.)

LAMONTAGNE, L., BÉLANGER, M., BERGER, (CRDV) - J., PELLETIER, P. (APG)

5. DATE DE PUBLICATION DU DOCUMENT (mois et année)	6a. NOMBRE DE PAGES	6b. NOMBRE DE RÉFÉRENCE
Mai 1998	51	7

7. DESCRIPTION DU DOCUMENT (La catégorie du document, par exemple rapport, note technique ou mémorandum. Indiquer les dates lorsque le rapport couvre une période définie.)

Note technique

8. PARRAIN (le nom et l'adresse)

CRDV

9a. NUMÉRO DU PROJET OU DE LA SUBVENTION (Spécifier si c'est un projet ou une subvention)	9b. NUMÉRO DE CONTRACT
Projet 3aa15	S/O

10a. NUMÉRO DU DOCUMENT DE L'ORGANISME EXPÉDITEUR	10b. AUTRES NUMÉROS DU DOCUMENT
CRDV TM-9738	N/A

11. ACCÈS AU DOCUMENT (Toutes les restrictions concernant une diffusion plus ample du document, autres que celles inhérentes à la cote de sécurité.)

- [X] Diffusion illimitée
- [ ] Diffusion limitée aux entrepreneurs des pays suivants (spécifier)
- [ ] Diffusion limitée aux entrepreneurs canadiens (avec une justification)
- [ ] Diffusion limitée aux organismes gouvernementaux (avec une justification)
- [ ] Diffusion limitée aux ministères de la défense
- [ ] Autres (préciser) :

12. ANNONCE DU DOCUMENT (Toutes les restrictions à l'annonce bibliographique de ce document. Cela correspond, en principe, aux données d'accès au document (11). Lorsqu'une diffusion supplémentaire (à d'autres organismes que ceux précisés à la case 11) est possible, on pourra élargir le cercle de diffusion de l'annonce.)

Illimitée.

SANS CLASSIFICATION

COTE DE LA SÉCURITÉ DE LA FORMULE

13. SOMMAIRE (Un résumé clair et concis du document. Les renseignements peuvent aussi figurer ailleurs dans le document. Il est souhaitable que le sommaire des documents classifiés soit non classifié. Il faut inscrire au commencement de chaque paragraphe du sommaire la cote de sécurité applicable aux renseignements qui s'y trouvent, à moins que le document lui-même soit non classifié. Se servir des lettres suivantes: (S), (C), (R) ou (NC). Il n'est pas nécessaire de fournir ici des sommaires dans les deux langues officielles à moins que le document soit bilingue.)

Bien que les systèmes de commandement et contrôle soient déployés dans des environnements non déterministes dont les conditions évoluent rapidement et de manière incertaine, peu de mécanismes adaptés à ces caractéristiques sont utilisés lors de leur conception. Le paradigme des "systèmes adaptatifs intelligents" (AIS) offre des solutions qui pourraient pallier ces lacunes; il permet, entre autres, d'adapter le comportement d'un système en réponse à des limitations de ressources, des échéances temporelles et d'autres facteurs contraignants liés à l'environnement. Dans ce document, on décrit les principes de l'un de ces paradigmes, le système de type "à raisonnement procédural" ("Procedural Reasoning System", PRS). On présente également l'évaluation de UM-PRS, un système développé par une équipe de chercheurs de l'Université du Michigan.

Even though command and control (C2) systems are deployed in non-deterministic environments with rapidely changing and uncertain conditions, very few mechanisms supporting these characteristics are incorporated in their design. The "Adaptive Intelligent System" (AIS) paradigm offers solutions that could enhance C3I systems response to these deficiencies. For example, AIS technologies enable the adaptation of system behavior to resource limitations, temporal deadlines and other limiting factors inherent to the environment. This document describes the principles of one of these paradigms, the Procedural Reasoning System (PRS). Furthermore, an evaluation of a PRS implementation, UM-PRS, a software package developed by a research team at the University of Michigan, is presented.

14. MOTS-CLÉS, DESCRIPTEURS OU RENSEIGNEMENTS SPÉCIAUX (Expressions ou mots significatifs du point de vue technique, qui caractérisent un document et peuvent aider à le cataloguer. Il faut choisir des termes qui n'exigent pas de cote de sécurité. Des renseignements tels que le modèle de l'équipement, la marque de fabrique, le nom de code du projet militaire, la situation géographique, peuvent servir de mots-clés. Si possible, on doit choisir des mots-clés d'un thésaurus, par exemple le "Thesaurus of Engineering and Scientific Terms (TESTS) ". Nommer ce thésaurus. Si l'on ne peut pas trouver de termes non classifiés, il faut indiquer la classification de chaque terme comme on le fait avec le titre.)

Français:

- système à raisonnement procédural
- PRS
- UM-PRS
- intelligence artificielle
- système intelligent adaptatif

Anglais:

- procedural reasoning system
- PRS
- UM-PRS
- Artificial Intelligence
- Adaptative

UNCLASSIFIED

Requests for documents
should be sent to:

DIRECTORATE RESEARCH AND DEVELOPMENT
INFORMATION MANAGEMENT
Dept. of National Defence
Ottawa, Ontario
K1A 0K2

Tel.: (613) 995-2971
Fax: (613) 996-0392

#508509

Toute demande de document
doit être adressée à:

DIRECTEUR-GESTION DE L'INFORMATION DE RECHERCHE
ET DE DÉVELOPPEMENT
Ministère de la Défense nationale
Ottawa, Ontario
K1A 0K2

Téléphone: (613) 995-2971
Télécopieur: (613) 996-0392

SANS CLASSIFICATION